Journal of Machine Learning Research 9 (2008) 2113-2139 Submitted 1/08; Published 10/08

Gradient Tree Boosting for Training Conditional Random Fields

Thomas G. Dietterich TGD@EECS.OREGONSTATE.EDU
Guohua Hao HAOG@EECS.OREGONSTATE.EDU
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331, USA

Adam Ashenfelter ASHENFAD@CLEVERSET.COM
Cleverset, Inc.
Corvallis, OR 97330, USA

Editor: Michael Collins

Abstract

Conditional random fields (CRFs) provide a flexible and powerful model for sequence labeling problems. However, existing learning algorithms are slow, particularly in problems with large numbers of potential input features and feature combinations. This paper describes a new algorithm for training CRFs via gradient tree boosting. In tree boosting, the CRF potential functions are represented as weighted sums of regression trees, which provide compact representations of feature interactions. So the algorithm does not explicitly consider the potentially large parameter space. As a result, gradient tree boosting scales linearly in the order of the Markov model and in the order of the feature interactions, rather than exponentially as in previous algorithms based on iterative scaling and gradient descent. Gradient tree boosting also makes it possible to use instance weighting (as in C4.5) and surrogate splitting (as in CART) to handle missing values. Experimental studies of the effectiveness of these two methods (as well as standard imputation and indicator feature methods) show that instance weighting is the best method in most cases when feature values are missing at random.

Keywords: sequential supervised learning, conditional random fields, functional gradient, gradient tree boosting, missing values

1. Introduction

Many applications of machine learning involve assigning labels collectively to sequences of objects. For example, in natural language processing, the task of part-of-speech (POS) tagging is to label each word in a sentence with a part of speech tag ("noun", "verb" etc.) (Ratnaparkhi, 1996). In computational biology, the task of protein secondary structure prediction is to assign a secondary structure class to each amino acid residue in the protein sequence (Qian and Sejnowski, 1988).

We call this class of problems *sequential supervised learning* (SSL), and it can be formulated as follows:

Given: A set of training examples of the form (X_i, Y_i), where each $X_i = (\mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,T_i})$ is a sequence of T_i feature vectors and each $Y_i = (y_{i,1}, \ldots, y_{i,T_i})$ is a corresponding sequence of class labels, where $y_{i,t} \in \{1, \ldots, K\}$.

Find: A classifier H that, given a new sequence X of feature vectors, predicts the corresponding sequence of class labels $Y = H(X)$ accurately.

Perhaps the most famous SSL problem is the NETtalk task of pronouncing English words by assigning a phoneme and stress to each letter of the word (Sejnowski and Rosenberg, 1987). Other applications of SSL arise in information extraction (McCallum et al., 2000) and handwritten word recognition (Taskar et al., 2004).

Early attempts to apply machine learning to SSL problems were based on *sliding windows*. To predict label y_t, a sliding window method uses features drawn from some "window" of the X sequence. For example, a 5-element window $w_t(X)$ would use the features $\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}$. Sliding windows convert the SSL problem into a standard supervised learning problem to which any ordinary machine learning algorithm can be applied. However, in most SSL problems, there are correlations among successive class labels y_t. For example, in part-of-speech tagging, adjectives tend to be followed by nouns. In protein sequences, alpha helixes and beta structures always involve multiple adjacent residues. These correlations can be exploited to increase classification accuracy.

The best-known method for capturing the $y_{t-1} \leftrightarrow y_t$ correlation is the hidden Markov model (HMM) (Rabiner, 1989), which is a generative model of $P(X,Y)$, the joint distribution of the observation sequence and label sequence. In this model, the joint distribution is factored as $P(X,Y) = \prod_t P(y_t|y_{t-1})P(\mathbf{x}_t|y_t)$, and the observation distribution is further factored as $P(\mathbf{x}_t|y_t) = \prod_j P(\mathbf{x}_{t,j}|y_t)$. This assumption of independence of each input feature $\mathbf{x}_{t,j}$ conditioned on y_t makes HMMs unable to model arbitrary, non-independent input features, and this limits the accuracy and "engineerability" of HMMs.

Recent research has instead focused on discriminative models, in which arbitrary and non-independent observation features can be easily incorporated. Much machine learning research has shown that discriminative models tend to be more accurate and more robust to incorrect modeling assumptions (Ng and Jordan, 2002). McCallum and his collaborators introduced maximum entropy Markov models (MEMMs) (McCallum et al., 2000) and conditional random fields (CRFs) (Lafferty et al., 2001). MEMMs are directed graphical models of the form $P(Y|X) = \prod_t P(y_t|y_{t-1}, w_t(X))$, where $w_t(X)$ is a sliding window over the X sequence. They are easy to train, but they suffer from the *label bias problem* that results from the local normalization at each time step t. Conditional random fields are undirected models of the form $P(Y|X) = 1/Z(X) \exp \sum_t \Psi(y_t, y_{t-1}, w_t(X))$, where $Z(X)$ is a global normalizing term and $\Psi(y_t, y_{t-1}, w_t(X))$ is a potential function that scores the compatibility of y_t, y_{t-1}, and $w_t(X)$. The global normalization avoids the label bias problem but makes training much more computationally expensive. CRFs have been applied to many problems with excellent results including POS tagging (Lafferty et al., 2001) and noun-phrase chunking (Sha and Pereira, 2003).

Kernel-based methods have also been extended to the SSL case. The hidden Markov SVM (Altun et al., 2003; Tsochantaridis et al., 2004) and max-margin Markov networks (Taskar et al., 2004) learn a discriminant function $F(X,Y')$ that assigns a real valued score to each possible label sequence Y' to maximize the margin between the correct label sequence Y and all competing incorrect label sequences.

Training CRFs is difficult for several reasons. First, as with all collective classification problems, training requires performing inference. In particular, all algorithms must compute the conditional log likelihood $\log P(Y_i|X_i)$ for each training example (X_i, Y_i) in each iteration. This is expensive, and it dictates that training algorithms should try to minimize the number of iterations and maximize the amount of progress made in each iteration. Second, in many SSL applications, the space of potential

features for describing the arguments of ψ (i.e., y_t, y_{t-1}, and $w_t(X)$) is immense. Even in the simple case where ψ is represented as a simple linear function $W \cdot F(y_t, y_{t-1}, w_t(X))$, there can be millions of weights to learn in W. In POS tagging and semantic role labeling, for example, it is common to have one feature (and hence, one weight) for every combination of a word and a pair of class labels. Furthermore, in most applications, performance is improved if the algorithm can consider combinations of these basic features (e.g., word n-grams, feature conjunctions and disjunctions, etc.). If feature interactions are permitted, the number of parameters to be learned explodes. Finally, in some problems, feature values can be missing, and this is difficult for discriminative training algorithms to handle.

There has been steady progress in algorithms for training CRFs. The initial paper (Lafferty et al., 2001) introduced an iterative scaling algorithm, which was reported to be exceedingly slow. Several groups have implemented gradient ascent methods (such as Sha and Pereira, 2003), but naive implementations are also very slow. McCallum's Mallet system (McCallum, 2002) employs the BFGS algorithm, which is an approximate second order method, to speed up the training of CRFs and improve the prediction accuracy. More recently, Vishwanathan et al. (2006) proposed to use stochastic gradient method to train CRFs, and accelerate this process via the Stochastic Meta-Descent (SMD), which is a gain adaptation method. The resulting algorithm is much faster than the BFGS algorithm and scales well on large data sets.

In this paper, we introduce a different approach for training the potential functions based on Friedman's gradient tree boosting algorithm (Friedman, 2001). In this method, the potential functions are represented by sums of regression trees, which are grown stage-wise in the manner of Adaboost (Freund and Schapire, 1996). Because each iteration adds an entire regression tree to the potential function, each iteration can take a big step in parameter space, and hence, reduce the number of iterations needed. Tree boosting also addresses the problem of dealing with feature interactions. Each regression tree can be viewed as defining several new feature combinations—one corresponding to each path in the tree from the root to a leaf. The resulting potential functions still have the form of a linear combination of features, but the features can be quite complex. Another advantage of tree boosting is that it is able to handle missing values in the inputs using clever methods specific to regression trees, such as the instance weighting method of C4.5 (Quinlan, 1993) and the surrogate splitting method of CART (Breiman et al., 1984). Finally, the algorithm is fast and straightforward to implement. In addition, there may be some tendency to avoid overfitting because of the "ensemble effect" of combining multiple regression trees.

This paper describes the gradient tree boosting algorithm including methods for incorporating weight penalties into the procedure. It then compares training time and generalization performance against McCallum's Mallet system. The results show that our implementation of tree boosting is competitive with Mallet in both speed and accuracy and that additional improvements in our implementation of the forward-backward algorithm would likely produce a system that is faster than both systems. We also perform experiments to evaluate the effectiveness of four methods for handling missing values (instance weighting, surrogate splits, indicator features, and imputation). The results show that instance weighting works best, but that imputation also works surprisingly well.

This leads to two conclusions. First, for CRF models, instance weighting combined with gradient tree boosting can be recommended as a good algorithm for learning in the presence of missing values. Second, for all SSL methods, imputation can be employed to provide a reasonable missing values method.

2. Conditional Random Fields

Let (X,Y) be a sequential labeled training example, where $X = (\mathbf{x}_1,\ldots,\mathbf{x}_T)$ is the observation sequence and $Y = (y_1,\ldots,y_T)$ is the sequence of labels, where $y_t \in \{1,\ldots,K\}$ for all t. A conditional random field is a linear chain Markov random field (Geman and Geman, 1984) over the label sequence Y globally conditioned on the observation sequence X. The probability distribution can be written as

$$P(Y|X) = \frac{1}{Z(X)} \exp\left[\sum_t \Psi_t(y_t,X) + \Psi_{t-1,t}(y_{t-1},y_t,X)\right] \ ,$$

where $\Psi_t(y_t,X)$ and $\Psi_{t-1,t}(y_{t-1},y_t,X)$ are *potential functions* that capture (respectively) the degree to which y_t is compatible with X and the degree to which y_t is compatible with a transition from y_{t-1} and with X. These potential functions can be arbitrary real-valued functions. The exponential function ensures that $P(Y|X)$ is positive, and the normalizing constant $Z(X) = \sum_{Y'} \exp[\sum_t \Psi_t(y'_t,X) + \Psi_{t-1,t}(y'_{t-1},y'_t,X)]$ ensures that $P(Y|X)$ sums to 1. If given sufficiently rich potential functions, this model can represent any first-order Markov distribution $P(Y|X)$ subject to the assumption that $P(Y|X) > 0$ for all X and Y (Besag, 1974; Hammersley and Clifford, 1971). Normally, it is assumed that the potential functions do not depend on t, and we will adopt this assumption in this paper.

To apply a CRF to an SSL problem, we must choose a representation for the potential functions. Lafferty et al. (2001) studied potential functions that are weighted combinations of binary features:

$$\begin{aligned} \Psi_t(y_t,X) &= \sum_a \beta_a g_a(y_t,X) \ , \\ \Psi_{t-1,t}(y_{t-1},y_t,X) &= \sum_b \lambda_b f_b(y_{t-1},y_t,X) \ , \end{aligned}$$

where the β_a's and λ_b's are trainable weights, and the features g_a and f_b are boolean functions. In part-of-speech tagging, for example, $g_{234}(y_t,X)$ might be 1 when $\mathbf{x}_t$ is the word "bank" and y_t is the class "noun" (and 0 otherwise). As with sliding window methods, it is natural to define features that depend only on a sliding window $w_t(X)$ of X values. This linear parameterization can be seen as an extension of logistic regression to the sequential case.

CRFs can be trained by maximizing the log likelihood of the training data, possibly with a regularization penalty to prevent overfitting. Let $\Theta = \{\beta_1,\ldots,\lambda_1,\ldots\}$ denote all of the tunable parameters in the model. Then we seek to maximize the objective function

$$\begin{aligned} J(\Theta) &= \log \prod_i P(Y_i \mid X_i) \\ &= \sum_i \log \frac{1}{Z(X_i)} \exp\left[\sum_t \Psi_t(y_{i,t},X_i) + \Psi_{t-1,t}(y_{i,t-1},y_{i,t},X_i)\right] \\ &= \sum_i \sum_t \Psi_t(y_{i,t},X_i) + \Psi_{t-1,t}(y_{i,t-1},y_{i,t},X_i) - \log Z(X_i) \\ &= \sum_i \sum_t \sum_a \beta_a g_a(y_{i,t},X_i) + \sum_b \lambda_b f_b(y_{i,t-1},y_{i,t},X_i) - \log Z(X_i) \ . \end{aligned}$$

A drawback of this linear parameterization is that it assumes that each feature makes an independent contribution to the potential functions. Of course it is possible to define more features to capture combinations of the basic features, but this leads to a combinatorial explosion in the number

of features, and hence, in the dimensionality of the optimization problem. For example, in protein secondary structure prediction, Qian and Sejnowski (1988) found that a 13-residue sliding window gave best results for neural network methods. There are $3^2 \times 13 \times 20 = 2340$ basic f_b features that can be defined over this window. If we consider fourth-order conjunctions of such features, we obtain more than 10^{12} features. This is obviously infeasible.

McCallum's Mallet system (McCallum, 2002) implements standard CRFs and CRFs with feature induction (McCallum, 2003). When feature induction is turned on, the learner starts with a single constant feature and (every 8 iterations) introduces new feature conjunctions by taking conjunctions of the basic features with features already in the model. Candidate conjunctions are evaluated according to their incremental impact on the objective function. He demonstrates significant improvements in speed and classification accuracy compared to a CRF that only includes the basic features. In this paper, we employ the gradient tree boosting method (Friedman, 2001) to construct complex features from the basic features as part of a stage-wise construction of the potential functions. The regression trees grown at each step are compact representations of complex features.

3. Gradient Tree Boosting

Suppose we wish to solve a standard supervised learning problem where the training examples have the form $(\mathbf{x}_i, y_i)$, $i = 1, \ldots, N$ and $y_i \in \{1, \ldots, K\}$. We wish to fit a model of the form

$$P(y \mid \mathbf{x}) = \frac{\exp \Psi(y, \mathbf{x})}{\sum_{y'} \exp \Psi(y', \mathbf{x})} \ .$$

Gradient tree boosting is based on the idea of *functional gradient ascent*. In ordinary gradient ascent, we would parameterize Ψ in some way, for example, as a linear function,

$$\Psi(y, \mathbf{x}) = \sum_a \beta_a g_a(y, \mathbf{x}) \ .$$

Let $\Theta = \{\beta_1, \ldots\}$ represent all of the tunable parameters in this function. In gradient ascent, the fitted parameter vector after iteration m, Θ_m, is a sum of an initial parameter vector Θ_0 and a series of gradient ascent steps δ_m:

$$\Theta_m = \Theta_0 + \delta_1 + \cdots + \delta_m \ ,$$

where each δ_m is computed as a step in the direction of the gradient of the log likelihood function:

$$\delta_m = \eta_m \nabla_\Theta \sum_i \log P(y_i \mid \mathbf{x}_i; \Theta) \Big|_{\Theta_{m-1}} \ ,$$

and η_m is a parameter that controls the step size.

Functional gradient ascent is a more general approach. Instead of assuming a linear parameterization for Ψ, it just assumes that Ψ will be represented by a weighted sum of functions:

$$\Psi_m = \Psi_0 + \Delta_1 + \cdots + \Delta_m \ .$$

Each Δ_m is computed as a *functional gradient*:

$$\Delta_m = \eta_m \, E_{\mathbf{x},y} \left[\nabla_\Psi \log P(y \mid \mathbf{x}; \Psi)|_{\Psi_{m-1}} \right] \ .$$

The functional gradient indicates how we would like the function Ψ_{m-1} to change in order to increase the true log likelihood (i.e., on all possible points $(\mathbf{x},y)$). Unfortunately, we do not know the joint distribution $P(\mathbf{x},y)$, so we cannot evaluate the expectation $E_{\mathbf{x},y}[\cdot]$. We do have a set of training examples sampled from this joint distribution, so we can compute the value of the functional gradient at each of our training data points:

$$\Delta_m(y_i,\mathbf{x}_i) = \nabla_\Psi \sum_i \log P(y_i \mid \mathbf{x}_i;\Psi)\Big|_{\Psi_{m-1}} \quad .$$

We can then use these point-wise functional gradients to define a set of *functional gradient training examples*, $((\mathbf{x}_i,y_i),\Delta_m(y_i,\mathbf{x}_i))$, and then train a function $h_m(y,\mathbf{x})$ so that it approximates $\Delta_m(y_i,\mathbf{x}_i)$. Specifically, we can fit a regression tree h_m to minimize

$$\sum_i [h_m(y_i,\mathbf{x}_i) - \Delta_m(y_i,\mathbf{x}_i)]^2 \quad .$$

We can then take a step in the direction of this fitted function:

$$\Psi_m = \Psi_{m-1} + \eta h_m \quad .$$

Although the fitted function h_m is not exactly the same as the desired Δ_m, it will point in the same general direction (assuming there are enough training examples). So ascent in the direction of h_m will approximate true functional gradient ascent.

A key thing to note about this approach is that it replaces the difficult problem of maximizing the log likelihood of the data by the much simpler problem of minimizing squared error on a set of training examples. Friedman (2001) suggests growing h_m via a best-first version of the CART algorithm (Breiman et al., 1984; Friedman et al., 2000) and stopping when the regression tree reaches a pre-set number of leaves L. The pseudo-code of this algorithm is shown in Table 1. Overfitting is controlled by tuning L (e.g., by internal cross-validation).

In our experience, using L to control overfitting is a blunt tool that is hard to calibrate. In this paper, we instead introduce shrinkage into the algorithm for growing regression trees by adding a quadratic weight penalty. For each leaf in the regression tree h_m, the quantity that we minimize is the squared error of the examples $((\mathbf{x}_i,y_i),\Delta_m(y_i,\mathbf{x}_i))$ falling into this leaf plus a quadratic penalty:

$$\sum_i (\Delta_m(y_i,\mathbf{x}_i) - \hat{\delta})^2 + \lambda\hat{\delta}^2 \quad ,$$

where $\hat{\delta}$ is the output of this leaf and $\lambda > 0$ controls the strength of the penalty. Differentiating the above objective function with respect to $\hat{\delta}$ shows that the minimum is achieved at

$$\hat{\delta} = \frac{\sum_i \Delta_m(y_i,\mathbf{x}_i)}{\lambda + N} \quad , \tag{1}$$

where N is the total number of examples falling into this leaf. This has the nice interpretation that λ is an equivalent number of training examples with target values of 0. So this shrinks the leaf values (learned weights) toward zero. With this method, we can select a large number for L (the maximum number of leaves in the regression tree), and use λ to give fine control of overfitting. The algorithm shown in Table 1 can be adapted by using Equation 1 in the computation of function OUTPUT and function SQUAREDERROR. Experimental results show that this new algorithm works better and is more efficient than the original best-first version of the CART algorithm.

```
FITREGRESSIONTREE(Data, L)
// Data = {(x_i, y_i) : i = 1,...,N, x_i = (x_i1,...,x_ip)}
// NodeQueue is a priority queue of tree nodes where the first node has the minimum SplitScore
Root := FINDBESTSPLITATTRIBUTE(Data, NodeQueue)
NumLeaves := 1
while ((NumLeaves < L) AND NOTEMPTY(NodeQueue))
    Node := REMOVEFRONT(NodeQueue)
    TrueData := examples in Node whose values of SplitFeature are true
    FalseData := examples in Node whose values of SplitFeature are false
    TrueChild := FINDBESTSPLITATTRIBUTE(TrueData, NodeQueue)
    FalseChild := FINDBESTSPLITATTRIBUTE(FalseData, NodeQueue)
    SETCHILDNODES (Node, TrueChild, FalseChild)
    NumLeaves := NumLeaves + 1
end
return Root
end FITREGRESSIONTREE

FINDBESTSPLITATTRIBUTE(Data, NodeQueue)
SplitScore := 0, SplitFeature := 0
for j from 1 to p
    TrueData := {(x_i, y_i) ∈ Data : x_ij = 1}
    FalseData := {(x_i, y_i) ∈ Data : x_ij = 0}
    Gain := SQUAREDERROR(TrueData) + SQUAREDERROR(FalseData) − SQUAREDERROR(Data)
    if Gain < SplitScore
        SplitScore := Gain, SplitFeature := j
    end
end
Node := MAKELEAF(OUTPUT(Data), Data, SplitFeature, SplitScore)
if SplitFeature ≥ 1
    INSERT(Node, NodeQueue)
end
return Node
end FINDBESTSPLITATTRIBUTE
```

Table 1: Best-first version of the CART algorithm.

4. Training CRFs with Gradient Tree Boosting

In principle, it is straightforward to apply functional gradient ascent to train CRFs. All we need to do is to represent and train $\Psi(y_t, X)$ and $\Psi(y_{t-1}, y_t, X)$ as weighted sums of regression trees. Let

$$F^{y_t}(y_{t-1}, X) = \Psi(y_t, X) + \Psi(y_{t-1}, y_t, X)$$

be a function that computes the "desirability" of label y_t given values for label y_{t-1} and the input features X. There are K such functions F^k, one for each class label k. With this definition, the CRF has the form

$$P(Y|X) = \frac{1}{Z(X)} \exp \sum_t F^{y_t}(y_{t-1}, X) \ .$$

We now compute the functional gradient of $\log P(Y|X)$ with respect to $F^{y_t}(y_{t-1}, X)$. To simplify the computation, we replace X by $w_t(X)$, which is a window into the sequence X centered at $\mathbf{x}_t$. We will further assume, without loss of generality, that each window is unique, so there is only one occurrence of $w_t(X)$ in each sequence X.

Proposition 1 *The functional gradient of* $\log P(Y|X)$ *with respect to* $F^v(u, w_d(X))$ *is*

$$\frac{\partial \log P(Y|X)}{\partial F^v(u, w_d(X))} = I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v \mid w_d(X)) \ ,$$

where $I(y_{d-1} = u, y_d = v)$ *is 1 if the transition* $u \to v$ *is observed from position* $d-1$ *to position* d *in the sequence* Y *and 0 otherwise, and where* $P(y_{d-1} = u, y_d = v \mid w_d(X))$ *is the predicted probability of this transition according to the current potential functions.*

To demonstrate this proposition, we must first introduce the forward-backward algorithm for computing the normalizing constant $Z(X)$. We will assume that y_t takes the value $\perp$ for $t < 1$. Define the forward recursion by

$$\begin{aligned} \alpha(k,1) &= \exp F^k(\perp, w_1(X)) \\ \alpha(k,t) &= \sum_{k'} \exp F^k(k', w_t(X)) \cdot \alpha(k', t-1) \ , \end{aligned}$$

and the backward recursion by

$$\begin{aligned} \beta(k,T) &= 1 \\ \beta(k,t) &= \sum_{k'} \exp F^{k'}(k, w_{t+1}(X)) \cdot \beta(k', t+1) \ . \end{aligned}$$

The variables k and k' iterate over the possible class labels. The normalizer $Z(X)$ can be computed at any position t as

$$Z(X) = \sum_k \alpha(k,t)\beta(k,t) \ .$$

If we unroll the α recursion one step, we can also write this as

$$Z(X) = \sum_k \left[\sum_{k'} \alpha(k', t-1) \cdot \left[\exp F^k(k', w_t(X)) \right] \right] \beta(k,t) \ .$$

Table 2 shows the derivation of the functional gradient. In Equation 2, exactly one of the $F^{y_t}(y_{t-1}, w_t(X))$ terms will match $F^v(u, w_d(X))$, because $w_d(X)$ is unique. This term will have a derivative of 1, so we represent this by the indicator function $I(y_{d-1} = u, y_d = v)$. In Equation 3, we expand $Z(X)$ at position d using the forward-backward algorithm. Again because $w_d(X)$ is unique, only the product where $k' = u$ and $k = v$ will give a non-zero derivative, so this gives us Equation 4. The right-hand expression in Equation 4 is precisely the joint probability that $y_{d-1} = u$ and $y_d = v$ given X. **Q.E.D.**

If $w_d(X)$ occurs more than once in X, each match contributes separately to the functional gradient.

This functional gradient has a very satisfying interpretation: It is our error on a probability scale. If the transition $u \to v$ is observed in the training example, then the predicted probability $P(u, v \mid X)$

$$
\begin{aligned}
&\frac{\partial \log P(Y|X)}{\partial F^v(u, w_d(X))} \\
&= \frac{\partial}{\partial F^v(u, w_d(X))} \sum_t F^{y_t}(y_{t-1}, w_t(X)) - \log Z(X) \\
&= I(y_{d-1} = u, y_d = v) - \frac{\partial \log Z(X)}{\partial F^v(u, w_d(X))} \qquad (2) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial Z(X)}{\partial F^v(u, w_d(X))} \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \frac{\partial}{\partial F^v(u, w_d(X))} \sum_k \left[\sum_{k'} \left[\exp F^k(k', w_d(X)) \right] \cdot \alpha(k', d-1) \right] \beta(k, d) \qquad (3) \\
&= I(y_{d-1} = u, y_d = v) - \frac{1}{Z(X)} \left[\exp F^v(u, w_d(X)) \right] \alpha(u, d-1) \beta(v, d) \qquad (4) \\
&= I(y_{d-1} = u, y_d = v) - P(y_{d-1} = u, y_d = v \mid X)
\end{aligned}
$$

Table 2: Derivation of the functional gradient.

should be 1 in order to maximize the likelihood. If the transition is not observed, then the predicted probability should be 0. Functional gradient ascent simply involves fitting regression trees to these residuals.

The pseudo code for our gradient tree boosting algorithm is shown in Table 3. The potential function for each class k is initialized to zero. Then M iterations of boosting are executed. In each iteration, for each class k, a set $S(k)$ of functional gradient training examples is generated. Each example consists of a window $w_t(X_i)$ on the input sequence, a possible class label k' at time $t-1$, and the target Δ value. A regression tree having at most L leaves is fit to these training examples to produce the function $h_m(k)$. This function is then added to the previous potential function to produce the next function. In other words, we are setting the step size $\eta_m = 1$. We experimented with performing a line search at this point to optimize η_m, but this is very expensive. So we rely on the "self-correcting" property of tree boosting to correct any overshoot or undershoot on the next iteration.

The sets of generated examples $S(k)$ can become very large. For example, if we have 3 classes and 100 training sequences of length 200, then the number of training examples for each class k is $3 \times 100 \times 200 = 60,000$. Although regression tree algorithms are very fast, they still must consider all of the training examples! Friedman (2001) suggests two tricks for speeding up the computation: sampling and influence trimming. In sampling, a random sample of the training data is used for training. In influence trimming, data points with Δ values close to zero are ignored. We did not apply either of these techniques in our experiments.

The most related work to ours is the virtual evidence boosting (VEB) algorithm developed by Liao et al. (2007) for training CRFs. Both VEB and our approach use boosting for feature induction. However, VEB is a "soft" version of maximum pseudo-likelihood training, where the observed values of neighborhood labels are not used, but the probability distribution over neighborhood labels is used as virtual evidence. Our approach is a true maximum log likelihood method that does not depend on the pseudo-likelihood approximation. Another difference is that VEB only uses decision stumps to induce simple features, while our approach uses regression trees to induce more complex feature combinations.

TREEBOOST($Data, L$)
// $Data = \{(X_i, Y_i) : i = 1, \ldots, N\}$
for each class k, initialize $F_0^k(\cdot,\cdot) = 0$
for $m = 1, \ldots, M$
  for class k from 1 to K
    $S(k) :=$ GENERATEEXAMPLES($k, Data, Pot_{m-1}$)
      // where $Pot_{m-1} = \{F_{m-1}^u : u = 1, \ldots K\}$)
    $h_m(k) :=$ FITREGRESSIONTREE($S(k), L$)
    $F_m^k := F_{m-1}^k + h_m(k)$
  end
end
return F_M^k for all k
end TREEBOOST

GENERATEEXAMPLES($k, Data, Pot_m$)
$S := \{\}$
for example i from 1 to N
  execute the forward-backward algorithm on (X_i, Y_i)
    to get $\alpha(k,t)$ and $\beta(k,t)$ for all k and t
  for t from 1 to T_i
    for k' from 1 to K
      $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i) :=$
        $\dfrac{\alpha(k', t-1)\exp[F_m^k(k', w_t(X_i))]\beta(k,t)}{Z(X_i)}$
      $\Delta(k, k', i, t) := I(y_{i,t-1} = k', y_{i,t} = k) -$
        $P(y_{i,t-1} = k', y_{i,t} = k \mid X_i)$
      insert $((w_t(X_i), k'), \Delta(k, k', i, t))$ into S
    end
  end
end
return S
end GENERATEEXAMPLES

Table 3: Gradient tree boosting algorithm for CRFs.

5. Inference in CRFs

Once a CRF model has been trained, there are (at least) two possible ways to define a classifier $Y = H(X)$ for making predictions. First, we can predict the *entire sequence* Y that has the highest probability:

$$H(X) = \operatorname*{argmax}_Y P(Y|X) \ .$$

This makes sense in applications, such as part-of-speech tagging, where the goal is to make a coherent sequential prediction. This can be computed by the Viterbi algorithm (Rabiner, 1989), which has the advantage that it does not need to compute the normalizer $Z(X)$.

The second way to make predictions is to individually predict each y_t according to

$$H_t(X) = \operatorname*{argmax}_v P(y_t = v|X) \ ,$$

and then concatenate these individual predictions to obtain $H(X)$. This makes sense in applications where the goal is to maximize the number of individual y_t's correctly predicted, even if the resulting predicted sequence Y is incoherent. For example, a predicted sequence of parts of speech might not be grammatically legal, and yet it might maximize the number of individual words correctly classified. $P(y_t|X)$ can be computed by executing the forward-backward algorithm as

$$P(y_t|X) = \frac{\alpha(y_t,t)\beta(y_t,t)}{Z(X)} \ .$$

6. Handling Missing Values in CRFs with Gradient Tree Boosting

In some problem settings (e.g., activity recognition, sensor networks), the problem of missing values in the inputs can arise. The values of input features can be missing for a wide variety of reasons. Sensors may break or the sensor data feed may be lost or corrupted. Alternatively, input observations may not have been measured in all cases because, for example, they are expensive to obtain. Many methods for handling missing values have been developed for standard supervised learning, but many of them have not been tested on SSL problems. Recently, Sutton et al. (2006) used feature bagging method to deal with SSL problems where highly indicative features may be missing in the test data. A single CRF trained on all the features will be less robust, because the weights of weaker features will be undertrained. Feature bagging method divides all the original features into a collection of complementary and possibly overlapped feature subsets. Separate CRFs are trained on each subset and then combined.

With gradient tree boosting, a CRF is represented as a forest of regression trees. There exist very good methods for handing missing values when growing regression trees, which include instance weighting method of C4.5 (Quinlan, 1993) and surrogate splitting of CART (Breiman et al., 1984). An advantage of training CRFs with gradient tree boosting is that these missing values methods can be used directly in the process of generating regression trees over the functional gradient training examples.

6.1 Instance Weighting

The instance weighting method (Quinlan, 1993), also known as "proportional distribution", assigns a weight to each training example, and all splitting decisions are based on weighted statistics. Initially, each example has a weight of 1.0. When selecting a feature to split on, each boolean feature x_j is evaluated based on the expected weighted squared error of the split using only the training examples for which x_j is not missing. The best feature x_{j*} is chosen, and the training examples for which x_{j*} is not missing are sent to the appropriate child node. Suppose that n_{left} examples are sent to the left child and n_{right} examples are sent to the right child. The remaining training examples (i.e., those for which x_{j*} is missing) are sent to *both* children, but with reduced weight. The weight

of each example sent to the left child is multiplied by $n_{left}/(n_{left}+n_{right})$. Similarly, the weight of each example sent to the right child is multiplied by $n_{right}/(n_{left}+n_{right})$.

At test time, when the test example reaches the test on feature x_{j*}, if the feature value is present, then the example is routed left or right in the usual way. But if x_{j*} is missing, then the example is sent to both children (recursively). Let $\hat{y}_{left}$ be the predicted value computed by the left subtree and $\hat{y}_{right}$ be the predicted value computed by the right subtree. Then the value predicted by node $j*$ is the weighted average of these predictions:

$$\hat{y} = \frac{n_{left}\hat{y}_{left} + n_{right}\hat{y}_{right}}{n_{left}+n_{right}} \quad .$$

Instance weighting assumes that the training and test examples missing x_{j*} will on average behave exactly like the training examples for which x_{j*} is not missing.

6.2 Surrogate Splitting

The surrogate splitting method (Breiman et al., 1984) involves separate procedures during training and testing. During training, as the regression tree is being constructed (in the usual top-down, greedy way), the key step in the learning algorithm is to choose which feature to split on. Each boolean feature x_j is evaluated based only on the training examples that have non-missing values for that feature, and the best feature, x_{j*} is chosen. Each of the remaining features $j' \neq j*$ is then evaluated to determine how accurately it can predict the value of x_{j*}, and the features are sorted according to their predictive power. This sorted list of features, called the surrogate splits, is stored in the node.

At test time, when test example x is processed through the regression tree, if x_{j*} is not missing, then the example is processed as usual by sending it to the left child if x_{j*} is false and to the right child if x_{j*} is true. However if x_{j*} is missing, then surrogate split features are examined in order until a feature j' is found that is not missing. The value of this feature determines whether to branch left or right.

7. Experimental Results

We implemented gradient tree boosting algorithm for CRFs and compared it to McCallum's Mallet system (McCallum, 2002) on several data sets. We call our algorithm TREECRF. We use TREECRF-FB for the TREECRF with forward-backward predictions and TREECRF-V for the TREECRF with Viterbi predictions. MALLET denotes the Mallet package with McCallum's feature induction algorithm (McCallum, 2003) turned on. Similarly, we use MALLET-FB and MALLET-V for the MALLET with forward-backward predictions and Viterbi predictions respectively. We also used the Mallet package to train standard CRFs without feature induction. We call it BASELINE, which serves as the baseline method. As before, BASELINE-FB denotes BASELINE with forward-backward predictions and BASELINE-V denotes BASELINE with Viterbi predictions. Note that MALLET-FB algorithm and BASELINE-FB algorithm are not implemented in the original Mallet package. Instead we implemented them ourselves.

TREECRF, MALLET and BASELINE have parameters that must be set by the user. For all these algorithms, the user must set (a) the window size, (b) the order of the Markov model, which is set to be 1 in our experiments, and (c) the number of iterations to train. For TREECRF, the only

additional parameter is either the maximum number of leaves L in the regression trees using the best-first version of CART, or the regularization constant λ for the shrinkage alternative. For MALLET, the parameters are (a) the regularization penalty for squared weights (called the variance), (b) the number of iterations between feature inductions (kept constant at 8), (c) the number of features to add per feature induction (kept constant at 500), (d) the true label probability threshold (kept constant at 0.95), (e) the training proportions (kept constant at 0.2, 0.5, and 0.8). For BASELINE, the only additional parameter is the variance as in MALLET. Except for the variance, we kept all of MALLET's parameters fixed at the values recommended by Andrew McCallum (personal communication). We did not optimize the window size, but instead employed values that have been used in previous studies. The chosen sizes are given in the following section. To set the remaining parameters, we manually tried the following settings and chose the setting that gave the best internal cross-validation performance:

- Number of leaves in regression trees: 30, 50, 75, 100,
- TreeCRF regularization constant: 0, 5, 10, 20, 40, 80,
- Weight variance prior in Mallet package: 1, 5, 10, 20.

Throughout the experiments, we measured the performance by computing the prediction accuracy of individual labels, rather than individual sequences. McNemar's test is employed to assess the statistical significance of these results.

7.1 Data Sets

Protein Secondary Structure Benchmark (Qian and Sejnowski, 1988). Each observation sequence is a string of amino acid residues, and the corresponding output sequence is a string over the 3-letter alphabet $\{\alpha, \beta, \gamma\}$, where α indicates alpha helix, β indicates a beta sheet or beta turn, and γ indicates all other secondary structure types. There are 20 possible amino acid residues, and we represent each residue by a set of 20 indicator variables. There is a training set of 111 sequences and a test set of 17 sequences. An 11-residue sliding window is used in our experiments.

NETtalk Data Set. The original NETtalk task (Sejnowski and Rosenberg, 1987) is to assign a combination of phoneme and stress to each letter of the word so that the word is pronounced correctly. However, there are 140 legal phone-stress combinations, which gives a very large label space. Neither TREECRF nor MALLET is sufficient enough to work with such a large label space. Hence, we chose to study only the problem of assigning one of five possible stress labels to each letter. The labels are '2' (strong stress), '1' (medium stress), '0' (light stress), '$<$' (unstressed consonant, center of syllable to the left), and '$>$' (unstressed consonant, center of syllable to the right).

Each input sequence is an English word, a string of letters over the 26 letter alphabet. Each input observation is represented by 26 boolean indicator variables. There are 1000 training words and 1000 test words in our standard training and test sets. We employed a window size of 13 (window width of 6).

Hyphenation Data Set. The hyphenation task is to insert hyphens into words at points where it is legal to break a word for a new line. This problem appears widely in many word processing programs. The input sequences are English words, encoded as for the NETtalk task. The output class label has only two values to indicate whether or not a hyphen may legally follow the current

	TREECRF-FB		TREECRF-V	
	Shrinkage	Original	Shrinkage	Original
Protein	64.52**	62.70	62.05***	59.20
NETtalk	85.18***	84.08	85.20***	84.18
Hyphen	92.20	92.20	91.76	92.07
FAQ ai-general	95.65	95.69	95.72	96.02***
FAQ ai-neural	99.02	98.97	99.20***	99.05
FAQ aix	94.00	94.02	95.26*	95.15

Table 4: Performance comparison of TREECRF with different regression tree fitting algorithms. Entries marked with one or more stars are statistically significantly better than the alternative method. Specifically, * means $p < 0.025$, ** means $p < 0.005$ and *** means $p < 0.001$ according to McNemar's test.

letter. We manually constructed a training set of 1951 words and a test set of 908 words. The input window size is set to be 6 (i.e., 3 letters on either side of the potential hyphen location).

Usenet FAQs Data Sets. Each of the FAQ data sets consists of Frequently Asked Questions files for a Usenet newsgroup (McCallum et al., 2000). The FAQs for each newsgroup are divided in separate files: ai-general has 7 files, ai-neural has 7 files, and aix has 5 files. Every line of an FAQ file is labeled as either part of the header, a question, an answer, or part of the tail. Hence, each $\mathbf{x}_t$ consists of a line in the FAQ file, and the corresponding $y_t \in \{\text{header, question, answer, tail}\}$. The measure of accuracy is the number of individual lines correctly classified. McCallum provided us with the definitions of 20 features for each line $\mathbf{x}_t$. We made a slight correction to one of the features, so our results are not directly comparable to his. The size of the sliding window used here is 1. For each newsgroup, performance was measured by leave-1-out cross-validation: the CRF was trained on all-but-one of the files and tested on the remaining file. This was repeated with each file, and the results averaged.

7.2 Performance of Shrinkage in Regression Tree Generation

To evaluate the effectiveness of shrinkage in the regression tree fitting algorithm, we fixed L, the maximum number of leaves in regression trees, to be 100, and applied internal cross-validation to choose the best regularization constant λ. For purposes of comparison, we also implemented the original best-first regression tree generation algorithm. Internal cross-validation was employed to select the best value for L.

We ran these two implementations of TREECRF on each data set. The best performance of both forward-backward predictions and Viterbi predictions are reported as percentages, as shown in Table 4. There are 12 pairs of comparisons (6 data sets with 2 prediction algorithms). In six of them, TREECRF with shrinkage does statistically better than TreeCRF without shrinkage. In five of them, the performance of these two versions of TREECRF is statistically indistinguishable. In only one of them, TREECRF without shrinkage does statistically better than TREECRF with shrinkage. Based on the results of these experiments, we decided to only employ TREECRF with shrinkage in the remaining experiments.

		Protein	NETtalk	Hyphen	FAQ ai-general	FAQ ai-neural	FAQ aix
Accuracy (%)	TREECRF	64.52*	85.20**	92.20**	**95.65** **	99.20**	95.26**
	MALLET	64.43*	**85.94****	92.10**	92.70	99.31*	95.28**
	BASELINE	62.44	82.81	88.86	92.70	99.41	94.04
Cumulative CPU Seconds	TREECRF	419.6	454.6	39.2	3921.9	2177.7	2636.1
	MALLET	786.9	941.4	66.4	484.1	237.2	125.5
	BASELINE	32.8	13.7	8.8	63.0	40.3	34.1
Iterations	TREECRF	142	169	58	214	84	158
	MALLET	123	167	69	188	181	150
	BASELINE	66	34	47	128–195	72–112	80–140

Table 5: Performance of TREECRF, MALLET, and BASELINE on each data set. Entries marked with one or more stars are statistically significant than BASELINE. Specifically, * means $p < 0.005$, ** means $p < 0.001$ according to McNemar's test. Bolded numbers indicate the statistically better prediction accuracy between TREECRF and MALLET. The BASELINE method stops training if the optimization of loss functions converges. So for each FAQ data set, different training set may have different number of training iterations. Here we gave out the range of number of training iterations for each FAQ data set.

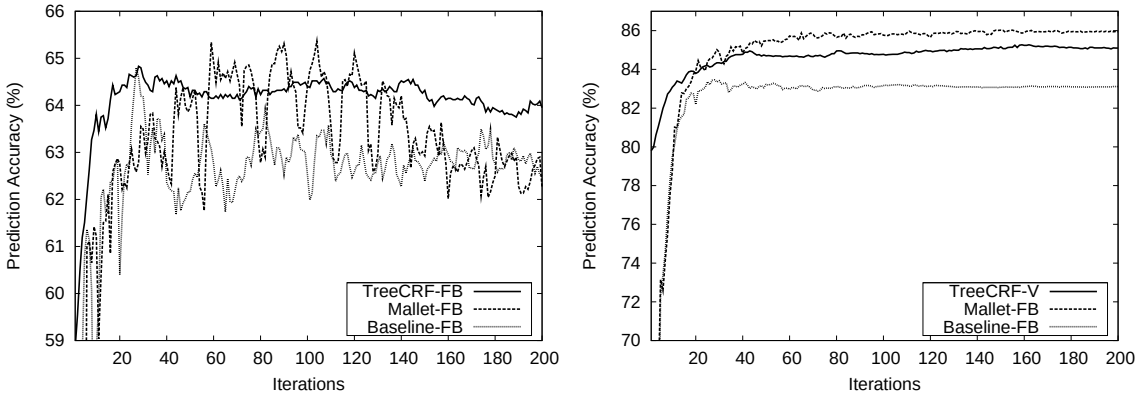
7.3 Comparison between TREECRF and MALLET

TREECRF and MALLET are the two leading CRF training methods that have feature induction capability. Here we compare the prediction accuracy and training speed of these two methods on each available data set. We also compare TREECRF and MALLET with the BASELINE method. For each method, internal cross-validation is applied to select the parameters that give the best performance of both forward-backward predictions and Viterbi predictions. The results reported here for each method are based on the prediction algorithm that gives higher prediction accuracy. All experiments were run on machines with 2.4 GHz Intel Xeon processors, 512KB cache, and 4GB memory.

Prediction Accuracy. Table 5 summarizes the prediction accuracy of TREECRF, MALLET, and BASELINE on each data set. McNemar's tests show that on four of the data sets, that is, protein, hyphen, FAQ ai-neural and FAQ aix, the difference between the prediction accuracy of TREECRF and MALLET is not statistically significant. On the FAQ ai-general data set, the prediction accuracy of TREECRF is statistically better than that of MALLET($p < 0.001$). Only on the NETtalk data set is the prediction accuracy of MALLET statistically better than that of TREECRF ($p < 0.05$). In comparison with the baseline method, the prediction accuracy of TREECRF and MALLET is statistically better than that of BASELINE in most cases. On the FAQ ai-general data set, the difference between MALLET and BASELINE is not statistically significant. Only on the FAQ ai-neural data set is the prediction accuracy of BASELINE statistically better than that of both TREECRF and MALLET.

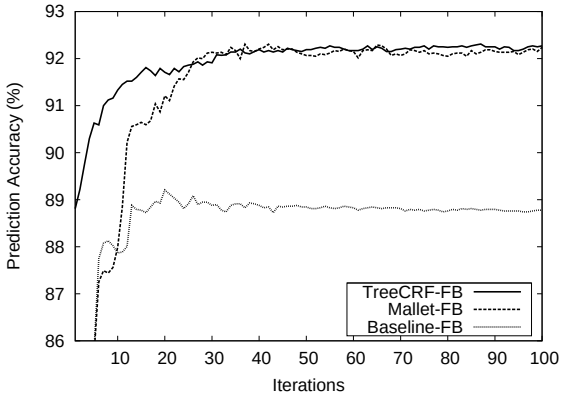
Figure 1 plots the prediction accuracy of TREECRF, MALLET and BASELINE as a function of the number of training iterations. One worrying aspect of MALLET is that the performance curve exhibits a high degree of fluctuation, which is clearly shown on Figure 1a, 1d, 1e and 1f. This is presumably due to the effect of introducing new features. But it also suggests that it will be difficult to find the optimal stopping points for avoiding overfitting.

Training Speed. It is difficult to directly compare the CPU time of these two methods, because TREECRF is written in C++ while MALLET is written in Java. However, comparing the CPU time

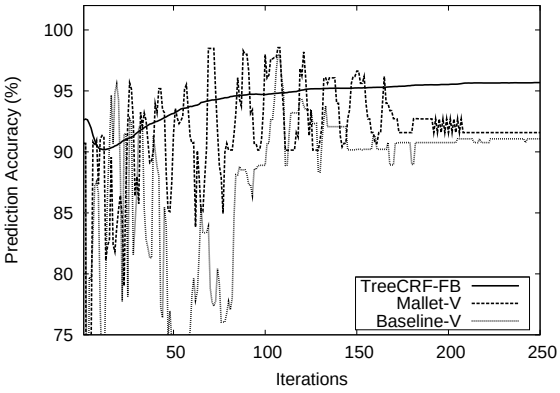


(a) Protein

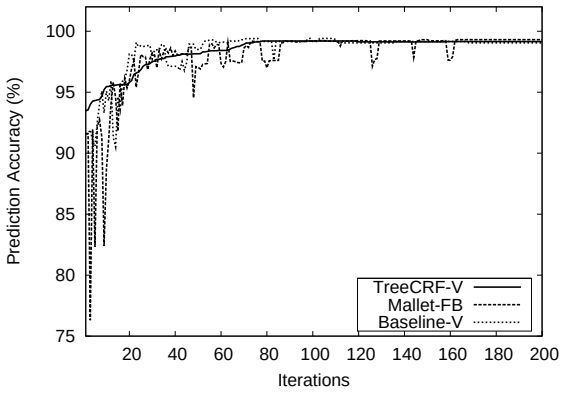
(b) NETtalk



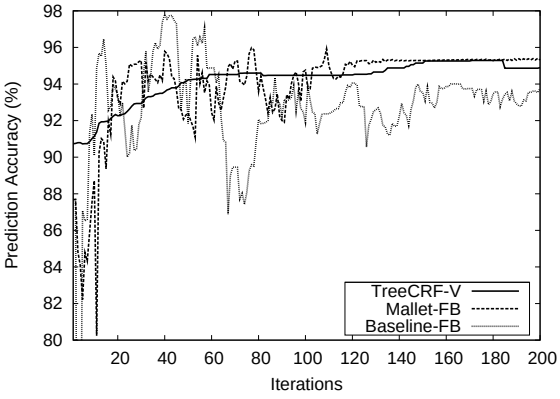
(c) Hyphen



(d) FAQ ai-general

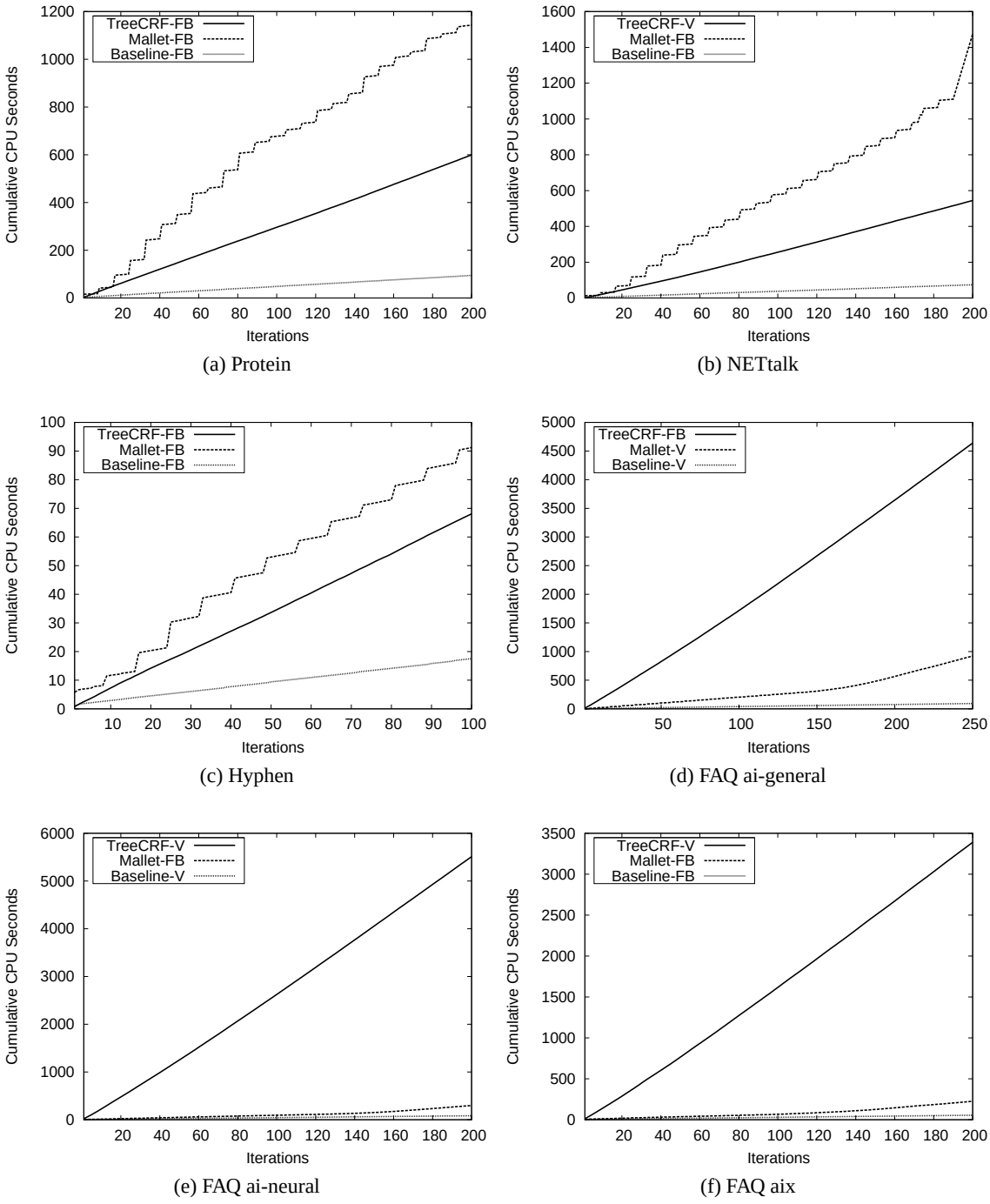


(e) FAQ ai-neural



(f) FAQ aix

Figure 1: Comparison of prediction accuracy on each data set.



(a) Protein (b) NETtalk

(c) Hyphen (d) FAQ ai-general

(e) FAQ ai-neural (f) FAQ aix

Figure 2: Comparison of cumulative CPU time on each data set.

Data Set	Average Length	Number of Features	Forward-Backward Seconds		Feature Induction Seconds	
			TREECRF	MALLET	TREECRF	MALLET
Protein	163	231	1.493	0.736	1.433	48.889
NETtalk	7	351	0.622	0.589	2.049	25.983
Hyphen	6	162	0.324	0.307	0.332	4.621
FAQ ai-general	1580	20	18.927	0.780	1.562	1.211
FAQ ai-neural	1832	20	26.998	0.526	1.894	1.656
FAQ aix	1806	20	16.658	0.352	1.199	1.123

Table 6: Comparison of average CPU seconds spent per iteration on forward-backward algorithm and feature induction algorithm in TREECRF and MALLET for each data set.

on different data sets can still give us some insight into the properties of these two methods. Figure 2 shows the number of cumulative CPU seconds consumed by these two methods on each data set. First, we can see that TREECRF scales linearly in the number of training iterations, because the cumulative CPU time has a constant slope. This makes sense, because for each potential function, only one regression tree is generated in each training iteration. Regression tree evaluations from previous iterations are cached so that they do not need to be re-evaluated. Without caching, the cumulative CPU curves for TREECRF would rise quadratically. Second, as shown in Figure 2a, 2b and 2c, TREECRF runs faster than MALLET on protein, NETtalk and hyphen data sets. But it is much slower than MALLET on FAQ data sets as shown in Figure 2d, 2e and 2f. The actual time required for each method to reach its peak performance on each data set is given in Table 5. Again we see that on the protein, NETtalk, and hyphen data sets, the time required for MALLET to reach its peak performance is about twice that of TREECRF. However, on the FAQ data sets, the time required for TREECRF to reach its peak performance is about 10-20 times more than for MALLET. BASELINE is faster than both TREECRF and MALLET as shown in Figure 2 and Table 5.

Analysis and Discussion. We can explain the training speed difference between TREECRF and MALLET by examining the details of these two methods. In both of them, most of the CPU time is spent on two major computations: forward-backward inference and feature induction/tree growing. The relative proportion of these two computations varies from problem to problem. To measure this, we instrumented both TREECRF and MALLET to track the amount of CPU time spent on each of these two computations. Table 6 shows that on domains with short sequences (Protein, NETtalk, and Hyphen), the time spent by both algorithms on forward-backward inference is about the same. But for domains with very long sequences, TREECRF consumes much more CPU time in forward-backward inference. Conversely, in domains with a small number of basic features (the FAQ data sets), the two methods consume roughly the same amount of CPU time in feature induction. But in domains with a large number of basic features, TREECRF is much more efficient than MALLET.

Why would the forward-backward cost of TREECRF be larger than for MALLET? TREECRF and MALLET use almost the same implementation of forward-backward algorithm except that in TREECRF the values of the potential functions at each position of the sequences are computed by evaluating the gradient regression trees generated in the current training iteration, while in MALLET those values are obtained by computing dot products of vectors, which is faster than tree evaluation. We hypothesize that the regression trees are more expensive to evaluate, not only because dot prod-

ucts are easier to compute than tree evaluations, but also possibly because of the reduced memory locality of regression trees.

Why would feature induction be more expensive in MALLET? In each feature induction iteration, MALLET considers conjoining all of the basic features to each of the existing compound features. Hence, if there are n basic features and C compound features, this costs nC. Furthermore, C grows over time, so the cost of feature induction gradually increases. In the cumulative CPU time plots of Figure 2, the "steps' in the "staircase" correspond to the feature induction iterations. In TREECRF, the cost of feature induction is the cost of growing a regression tree, which depends on the number of basic features n and the number of internal nodes in the tree L. This cost is nL, which remains constant across the iterations.

To verify our conjectures about the computational complexity of TREECRF and MALLET, we generated synthetic training data sets using a hidden Markov model (HMM) with 3 labels $\{l_1, l_2, l_3\}$ and 24 possible observations $\{o_1, \ldots, o_{24}\}$. To specify the observation distribution, for each label l_i, we randomly draw an observation from the set $\{o_{i*8-7}, \ldots, o_{i*8}\}$ with probability 0.6 and randomly draw an observation from the complement of this set with probability 0.4. The transition distribution is defined as $P(y_t = l_i \mid y_{t-1} = l_i) = 0.6$ and $P(y_t = l_j \mid y_{t-1} = l_i) = 0.2$ if $i \neq j$.

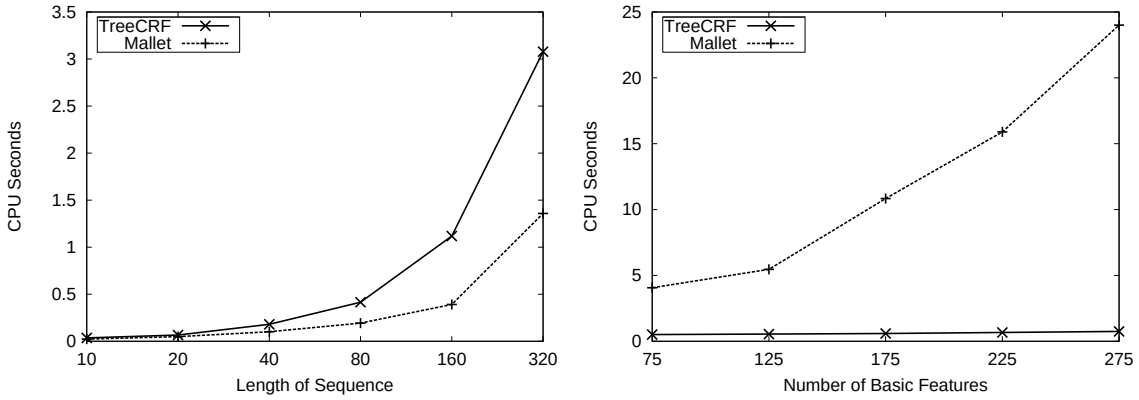
In order to measure the complexity of the forward-backward algorithm, we tried sequence lengths of 10, 20, 40, 80, 160 and 320. For each sequence length, we generated a training data set with 100 sequences and employed a sliding window of size 3. TREECRF and MALLET are run on each of these training data sets. Figure 3a shows the average CPU seconds spent per iteration on the forward-backward algorithm by these two methods. We see that the forward-backward algorithm in TREECRF implementation scales faster than that in MALLET implementation as the length of sequence increases.

In order to measure the complexity of the feature induction algorithms, we generated a training data set with 100 sequences. The length of each sequences is 100. We tried sliding window sizes of 3, 5, 7, 9 and 11, so that the number of input features at each sequence position takes the values of 75, 125, 175, 225 and 275 (because each input observation is represented by 25 boolean indicator variables). TREECRF and MALLET are run for each sliding window size. Figure 3b shows the average CPU seconds spent per iteration on the feature induction algorithm by these two methods. It is clear that the feature induction algorithm in MALLET spends more and more CPU time than that in TREECRF as the number of basic features increases. In all the experiments on synthetic data sets, TREECRF uses regression trees of maximum 100 leaves and shrinkage constant 40. MALLET uses weight variance prior 20.

This analysis suggests that the performance of TREECRF could be improved by "flattening" the ensemble of regression trees to compute the corresponding vector of features and vector of weights. Then the cost of potential function evaluations would be similar to that of MALLET, and we would have a method that was faster than both the current TREECRF and MALLET implementations.

7.4 Experimental Studies of Missing Values in TREECRF

We performed a series of experiments to evaluate the effectiveness of methods for handling missing values in TREECRF algorithm. In addition to the instance weighting and surrogate splitting methods described above, we also studied two simpler methods: imputation and indicator features. Let $x_{tj}, j = 1, \ldots, n$ be the input features describing a particular input observation $\mathbf{x}_t$. Imputation and indicator features are defined as follows:



(a) Forward-backward algorithm (b) Feature induction algorithm

Figure 3: Comparison of average CPU seconds spent per iteration on forward-backward algorithms and feature induction algorithms by TREECRF and MALLET.

Imputation: when a feature value x_{tj} is missing, it is replaced with the most common value for x_j in the training data among those feature values that are not missing. This strategy can be viewed as substituting the most likely value of x_j a priori or alternatively as substituting the value of x_j least likely to be informative.

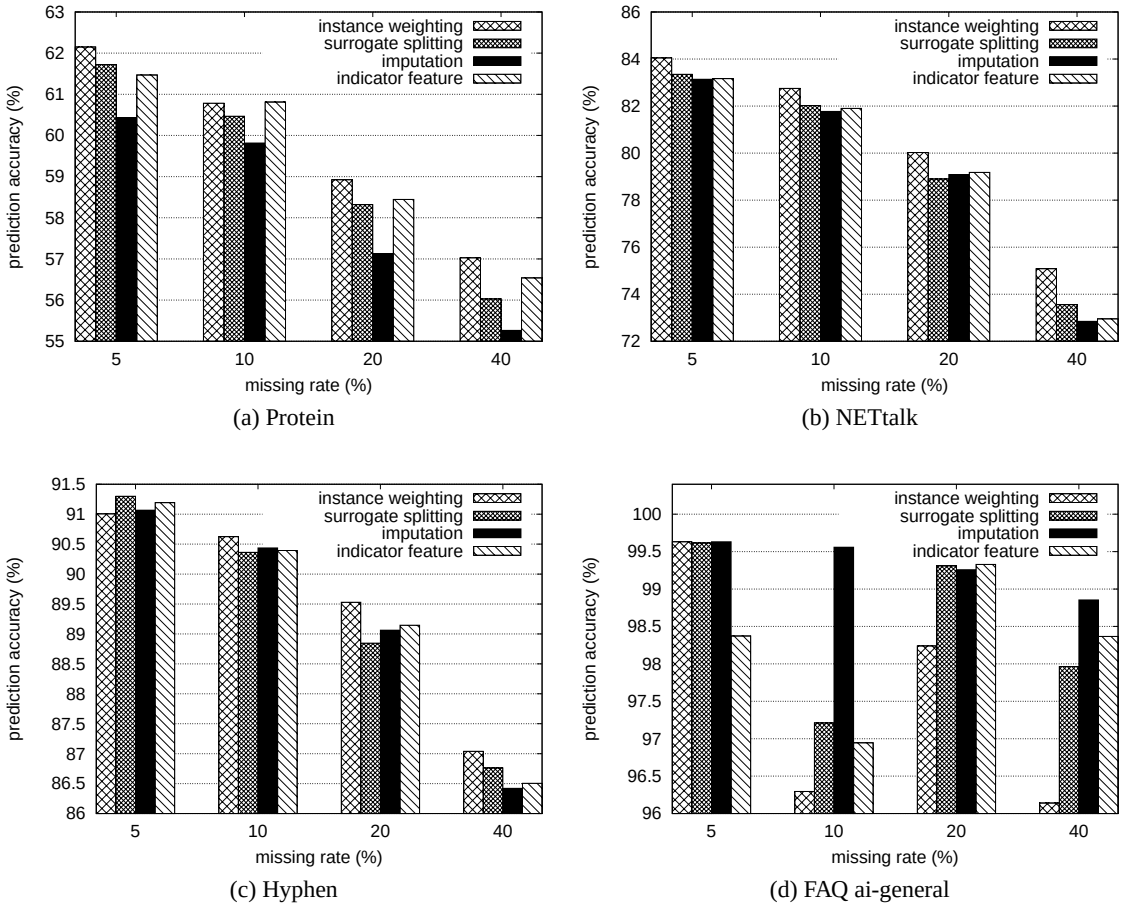
Indicator Features: a boolean feature $\tilde{x}_{tj}$ is introduced for each feature x_{tj} such that if x_{tj} is present, $\tilde{x}_{tj}$ is false. But if x_{tj} is missing, then $\tilde{x}_{tj}$ is true and x_{tj} is set to a fixed chosen value, typically 0. Indicator features make sense when the fact that a value is missing is itself informative. For example, if x_{tj} represents a temperature reading, it may be that extremely cold temperature values tend to be missing because of sensor failure.

We adopted a first-order Markov model in all the following experiments and employed an internal hold-out method to set the other parameters: Two-thirds of the original training set was used as sub-training set and the other one third was used as development set to choose parameter values. Final training was performed using the entire training set.

For each learning problem, we took the chosen training and test sets and inject missing values at rates of 5%, 10%, 20% and 40%. For a given missing rate, we generate five versions of the training set and five versions of the test set. A CRF is then trained on each of the training sets and evaluated on each of the test sets (for a total of 5 CRFs and 25 evaluations per missing rate). The label sequences are predicted by the forward-backward algorithm (i.e., we compute $\hat{y}_t = \text{argmax}_{y_t} P(y_t|X)$ for each t separately). Prediction accuracy is based on the number of individual labels correctly predicted in the label sequences. The final prediction accuracy is the average of all 25 cases.

To test the statistical significance of the differences among the four methods, we performed an analysis of deviance based on the generalized linear model discussed by Agresti (1996). We fit a logistic regression model

$$\log \frac{P(y_t = \hat{y}_t)}{1 - P(y_t = \hat{y}_t)} = \delta_1 m_1 + \delta_2 m_2 + \delta_3 m_3 + \sum_\ell \sigma_\ell S_\ell \ ,$$



(a) Protein (b) NETtalk

(c) Hyphen (d) FAQ ai-general

Figure 4: Performance of missing values methods for different missing rates.

where m_1, m_2, and m_3 are boolean indicator variables that specify which missing values method we are using and the S_ℓ's are indicator variables that specify which of the five training sets we are using. If $m_1 = m_2 = m_3 = 0$, then we are using instance weighting, which serves as our baseline method. If $m_1 = 1$, this indicates surrogate splitting, $m_2 = 1$ indicates imputation, and $m_3 = 1$ indicates the indicator feature method. Consequently, the fitted coefficients δ_1, δ_2, and δ_3 indicate the change in log odds (relative to the baseline) resulting from using each of these missing values methods. We can then test the hypothesis $\delta_i \neq 0$ against the null hypothesis $\delta_i = 0$ to determine whether missing values method i is different from the baseline method.

This statistical approach controls for variability due to the choice of the training set (through the σ_ℓ's) and variability due to the size of the test set.

Protein Secondary Structure Prediction. Figure 4a shows that instance weighting achieves the best prediction accuracy for each of the different missing rates. Table 7a shows that the base line missing values method, instance weighting, is statistically better than the other three missing values methods in most cases. In other cases, it is as good as other methods.

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	−0.018	−0.072*	−0.028*
10%	−0.013	−0.040*	0.001
20%	−0.025*	−0.074*	−0.020*
40%	−0.041*	−0.072*	−0.020*

(a) Protein

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	−0.051*	−0.066*	−0.064*
10%	−0.051*	−0.067*	−0.059*
20%	−0.069*	−0.057*	−0.052*
40%	−0.080*	−0.116*	−0.111*

(b) NETtalk

Missing rate	Surrogate splitting	Imputation	Indicator feature
5%	0.036*	0.007	0.023
10%	−0.031*	−0.022	−0.027*
20%	−0.071*	−0.049*	−0.040*
40%	−0.024*	−0.054*	−0.047*

(c) Hyphen

Missing rate	Instance weighting	Surrogate splitting	Indicator feature
5%	−8.824E−16	−0.043	−1.499*
10%	−2.161*	−1.867*	−1.961*
20%	−0.874*	0.072	0.100
40%	−1.243*	−0.584*	−0.359*

(d) FAQ ai-general

Table 7: Estimation of the coefficients corresponding to different missing values methods and statistical test results. In FAQ ai-general problem, imputation was the baseline method, so the coefficient values give the log odds of the change in accuracy relative to imputation. * means that the parameter value is statistically significantly different from zero ($p < 0.05$).

NETtalk Stress Prediction. In Figure 4b, we see that instance weighting does better than the other three missing values methods for all the different missing rates. The statistical tests reported in Table 7b show that the baseline method, instance weighting, is statistically better than each of the other missing value methods in all cases.

Hyphenation. Figure 4c shows that instance weighting is the best missing values method except for a missing rate of 5%. Statistical tests shown in Table 7c tell us that for missing rate of 5%, surrogate splitting is the best missing values method and the other three methods are not statistically significantly different from each other. For a missing rate of 10%, instance weighting and imputation are statistically better than the other two methods (and indistinguishable from each other). For missing rates of 20% and 40%, instance weighting is statistically better than the other three methods.

FAQ Document Segmentation. This task is based on the ai-general Usenet FAQ data set as we discussed before. We treat the first 6 files as the training set and the seventh file as the test set. The input window contains only the features corresponding to a single line in the file (window half-width of 0). Unlike in the previous data sets, instance weighting is no longer the best missing values method, as shown in Figure 4d. Instead, imputation performs very well for various missing value rates. Table 7d shows that imputation is statistically the best missing values method. For missing rates of 10% and 40%, it is statistically better than the other three methods. For a missing rate of 5%, it does as well as instance weighting and surrogate splitting. For a missing rate of 20%, it does as well as surrogate splitting and indicator features.

Analysis and Discussion. The four missing values methods are based on different assumptions about the input data. Imputation assumes that the most frequent value of a feature is the least informative and therefore presents the lowest risk of introducing errors into the learning process. Missing values are injected prior to converting the input features to binary. Hence, in the protein data set, missing values are introduced by choosing an amino acid residue position in the observa-

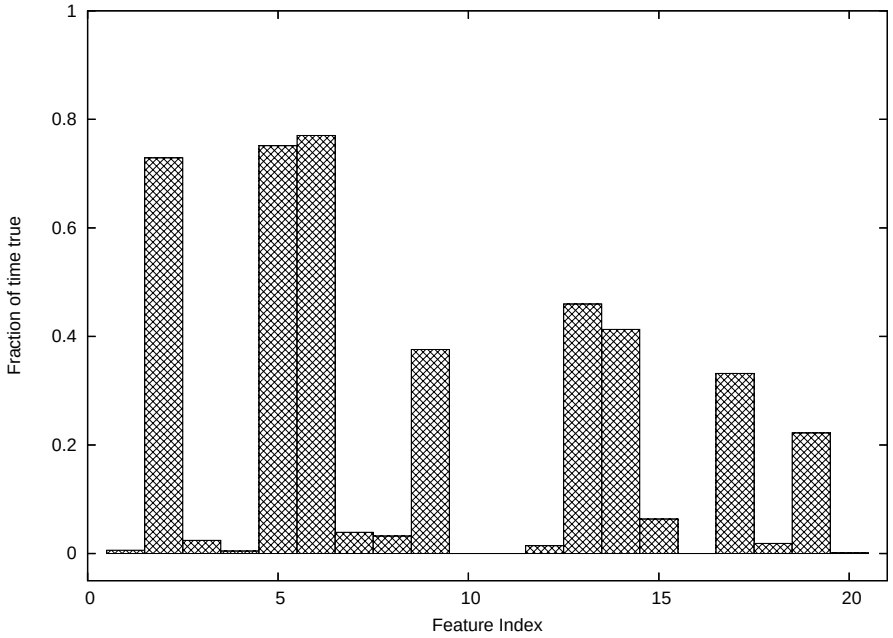


Figure 5: Fraction of the time that each FAQ feature is true (versus false). Features 1, 3, 4, 7, 8, 10, 11, 12, 16, 18, and 20 are rarely true.

tion sequence and setting all 20 boolean indicator features that represent that position to missing. Similarly, in the NETtalk and hyphenation problems, a letter is made to be missing by setting all 26 indicator features for that letter to missing. Similarly, imputation is computed at the amino acid or letter level, not at the level of boolean features. However, in the Usenet FAQ data set, since the binary features are not exclusive, imputation is computed at the level of boolean features. In the case of protein sequences, imputation will replace missing values with the most frequently-occurring amino acid, which is alanine, code 'A'. Alanine tends to form alpha helices, so this may cause the learning algorithms to over-predict the helix class, which may explain why imputation performed worst on the protein data set. In the case of English words, the most common letter is 'E', and it does not carry much information either about pronunciation or about hyphenation, so this may explain why imputation worked well in the NETtalk and hyphenation problems. Finally, in the ai-general FAQ data set, most of the features exhibit a highly skewed distribution, so that one feature value is much more common than another, as shown in Figure 5. Hence, in most cases, imputation with the most common feature value will supply the correct missing value. This may be why it worked best on that data set.

The indicator feature approach is based on the assumption that the presence or absence of a feature is meaningful (e.g., in medicine, a feature could be missing because a physician explicitly chose not to measure it). Because features were marked as missing completely at random, this is not true, so the indicator feature carries no positive information about the class label. However, in cases where imputation causes problems, the indicator feature approach may help prevent those problems by being more neutral. The learning algorithm can learn that if the indicator feature is set, then the actual feature value should be ignored. This may explain why the indicator feature method works slightly better in most cases than the imputation method.

The surrogate splitting method assumes that the input features are correlated with one another, so that if one feature is missing, its value can be computed from another feature. The protein, NETtalk, and hyphenation data sets have a single input feature for each amino acid or letter. Hence, if this input feature is missing, then there is no information about that position in the sequence. The only exception to this would be if there were strong correlations between successive amino acids or letters. However, such strong correlations do not exist much either in protein sequences or in English, with the possible exception of the letter 'q', which is always followed by 'u'. Note that the converse is not true: 'u' is not always preceded by 'q'. Based on these considerations, we would not expect surrogate splitting to work well in these domains, and it does not.

In the FAQ data set, each line is described by 20 features computed from the words in that line. In the experiment, each of these 20 features could be independently marked as missing, which is a bit unrealistic, since presumably the real missing values would involve some loss or corruption of the words making up the line, and this would affect multiple features. The 20 features do have some redundancy, so we would expect that surrogate splitting should work well, and it does for 5% and 20% missing rates.

The instance weighting method assumes that the feature values are missing at random and that the other features provide no redundant information, so the most sensible thing to do is to marginalize away the uncertainty about the missing values. Our experiments show that this is a very good strategy in all cases except for the FAQ data set, where the features are somewhat redundant.

8. Conclusions

In this paper, we presented TREECRF, a novel method for training conditional random fields based on gradient tree boosting. TREECRF has the ability to construct very complex feature conjunctions from basic features and scales much better than methods based on iterative scaling and simple gradient descent. It appears to match the L-BFGS algorithm implemented in MALLET, which also gives dramatic speedups when there are many potential features. In our experiments, TREECRF is as accurate as MALLET on four data sets, more accurate on one data set and less accurate on one data set. Its feature induction method is faster than that of MALLET for problems with a large number of features. But its forward-backward implementation is slower than that of MALLET for really long sequences. In addition, TREECRF is easier to implement and tune. It introduces only one tunable parameter (either the maximum number of leaves permitted in each regression tree or the regularization constant), whereas MALLET has many more parameters to consider. It is easier for the TREECRF to find the optimal stopping point to avoid overfitting, since its performance improves smoothly, while that of MALLET fluctuates wildly. Combining the benefit of these two methods will be a promising direction to pursue.

TREECRF also provides us with extra ability to handle missing data with instance weighting and surrogate splitting methods, which are not available in MALLET and other CRF training algorithms. The experiments suggest that when the feature values are missing at random, the instance weighting approach works very well. In the one domain where instance weighting did not work well, imputation was the best method. The indicator feature method was also very robust. The method of surrogate splitting was the most expensive method to run and the least accurate. Hence, we do not recommend using surrogate splits with conditional random fields. The good performance of the indicator features and imputation methods is encouraging, because these methods can be applied with all known methods for sequential supervised learning, not only with gradient tree

boosting. Since there is no one best method for handling missing values, as with many other aspects of machine learning, preliminary experiments on subsets of the training data are required to select the most appropriate method.

Acknowledgments

The authors would like to thank the anonymous reviewers and the action editor for their constructive input. We also gratefully acknowledge the support of NSF grants IIS-0083292 and IIS-0307592. Some of the material in this paper was first published at ICML-2004 (Dietterich et al., 2004).

References

Alan Agresti. *An Introduction to Categorical Data Analysis*. Wiley, New York, 1996.

Yasemin Altun, Ioannis Tsochantaridis, and Thomas Hofmann. Hidden markov support vector machines. In Tom Fawcett and Nina Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning (ICML 2003)*, pages 3–10. AAAI Press, 2003.

Julian Besag. Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society B*, 36(2):192–236, 1974.

Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth Publishing Company, 1984.

Thomas G. Dietterich, Adam Ashenfelter, and Yaroslav Bulatov. Training conditional random fields via gradient tree boosting. In *Proceedings of the 21st International Conference on Machine Learning (ICML 2004)*, pages 217–224, Banff, Canada, 2004. ACM Press.

Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning (ICML 1996)*, pages 148–156. Morgan Kaufmann, 1996.

Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 38(2):337–374, 2000.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6): 721–741, Nov. 1984.

John M. Hammersley and Peter Clifford. Markov fields on finite graphs and lattices. Technical report, Unpublished, 1971.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning (ICML 2001)*, pages 282–289. Morgan Kaufmann, 2001.

Lin Liao, Tanzeem Choudhury, Dieter Fox, and Henry A. Kautz. Training conditional random fields using virtual evidence boosting. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2530–2535, Hyderabad, India, January 6-12 2007.

Andrew McCallum. Efficiently inducing features of conditional random fields. In Christopher Meek and Uffe Kjaerulff, editors, *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003)*, pages 403–410. Morgan Kaufmann, 2003.

Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. Maximum entropy markov models for information extraction and segmentation. In *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pages 591–598. Morgan Kaufmann, 2000.

Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

Andrew Y. Ng and Michael Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

Ning Qian and Terrence J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA, 1993.

Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In Eric Brill and Kenneth Church, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 133–142, Somerset, New Jersey, 1996. Association for Computational Linguistics.

Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.

Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In Marti Hearst and Mari Ostendorf, editors, *HLT-NAACL 2003: Main Proceedings*, pages 213–220, Edmonton, Alberta, Canada, May 27 – June 1 2003. Association for Computational Linguistics.

Charles Sutton, Michael Sindelar, and Andrew McCallum. Reducing weight undertraining in structured discriminative learning. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 89–95, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin markov networks. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 25–32. MIT Press, Cambridge, MA, 2004.

Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the 21st International Conference on Machine Learning (ICML 2004)*, pages 823–830, Banff, Canada, 2004. ACM Press.

S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine learning (ICML 2006)*, pages 969–976, New York, NY, USA, 2006. ACM.